Gapped Suffix Arrays: a New Index Structure for Fast Approximate Matching

Maxime Crochemore, German Tischler

Gapped Suffix Arrays: a New Index Structure for Fast Approximate Matching

Maxime Crochemore[1,2], German Tischler[1,3]

[1] Dept. of Computer Science, King's College London, London WC2R 2LS, UK
{maxime.crochemore,german.tischler}@kcl.ac.uk
[2] Université Paris-Est, France
[3] Newton Fellow

Abstract. Approximate searching using an index is an important application in many fields. In this paper we introduce a new data structure called the gapped suffix array for approximate searching in the Hamming distance model. Building on the well known filtration approach for approximate searching, the use of the gapped suffix array can improve search speed by avoiding the merging of position lists.

1 Introduction

Pattern matching in textual data is a much studied question in Computer Science and large parts of books on algorithms on strings and sequences are devoted to the question (see e.g. [2]). Several types of applications require approximate matching rather than exact matching of the patterns. This is typically the situation for motif search and inference in biological molecular sequences because they allow some diversity without altering the basic information they carry. But this is not by far the only domain demanding approximate matching solutions. A main technique to deal with the question is the notion of alignment, which admits a considerable number of variants and is parameterised by the costs of allowed elementary operations (see e.g. [1]).

However there are actually two sub-problems depending on which are known first, patterns to be searched for or data to be searched. They admit totally different types of solutions. The paradigm solution for searching for approximate occurrences of a fixed pattern under the notion of Levenshtein operations is due to Landau and Vishkin [7], and the same authors designed a simpler version when only mismatches are considered [6]. The second type of solution appears when the data is to be searched for multiple patterns. It is then appropriate to index the data for accelerating their future inspection and analysis.

Indexing for approximate searches is the problem we address in the article, but with one important restriction: patterns are of fixed size. Moreover the proposed solution accommodates only few mismatches to be feasible with reasonable resources.

In this paper we introduce what we call the gapped Suffix Array. It is a data structure enhancing the standard suffix array and tailored to accept searches for patterns up to some mismatches.

2 Definitions

Let throughout this paper Σ be a finite ordered alphabet and let Σ^* denote the set of all finite strings over Σ. We denote the empty string by ϵ and the length of a string u by $|u|$. Let $y = y[0] \dots y[n-1]$ denote a string of finite length n over Σ which we call the text. We denote the factor starting at position i and ending at position j of some string x by $x[i\,..\,j]$. It is defined by $x[i\,..\,j] = x[\max(0,i)]x[\max(0,i)+1] \dots x[\min(j,|x|-1)]$ for $\max(0,i) \leq \min(j,|x|-1)$ and $x[i\,..\,j] = \epsilon$ otherwise. A prefix of a string x is $x[0\,..\,i]$ for any position i and a suffix of x is $x[i\,..\,|x|-1]$ for any position i. A string $u \in \Sigma^*$ is lexicographically smaller than a string $v \in \Sigma^*$ (which we denote by $u < v$), if $u \neq v$ and either $u = \epsilon$ or $u \neq \epsilon \neq v$ and $u[0] < v[0]$ or $u \neq \epsilon \neq v$ and $u[0] = u[0]$ and $u[1\,..\,|u|-1] < v[1\,..\,|v|-1]$. The array SA of length n is defined by $\mathsf{SA}[r]$ being the start position of the r'th lexicographically smallest non empty suffix of y, i.e. we obtain the relation

$$y[\mathsf{SA}[0]\,..\,n-1] < y[\mathsf{SA}[1]\,..\,n-1] < \ldots < y[\mathsf{SA}[n-1]\,..\,n-1] \ .$$

The array SA can be computed from the string y in linear time if $|\Sigma| \in O(n^c)$ for some constant c (in particular for $c = 0$, i.e. alphabets of constant size, cf. [2]). Let the array ISA be defined by $\mathsf{ISA}[\mathsf{SA}[r]] = r$ for $0 \leq r < n$. The length of the longest common prefix of $u, v \in \Sigma^*$, which we denote by is $\mathsf{lcp}(u,v)$, is the largest $l \leq \max(|u|,|v|)$ such that $u[0\,..\,l-1] = v[0\,..\,l-1]$. We denote the length of the longest common prefix of the two suffixes starting at position i and j of y by $\mathsf{lcp}_{\mathrm{y}}(i,j)$ and we define $\mathsf{lcp}_{\mathrm{r}}(r,q) = \mathsf{lcp}_{\mathrm{y}}(\mathsf{SA}[r],\mathsf{SA}[q])$ for $0 \leq r,q < n$. We define the LCP array by $\mathsf{LCP}[r] = \mathsf{lcp}_{\mathrm{r}}(r-1,r)$ for $1 \leq r < n$ and by $\mathsf{LCP}[0] = 0$. It is well known that the identity

$$\mathsf{lcp}_{\mathrm{r}}(r,q) = \min\{\mathsf{LCP}[r+1], \mathsf{LCP}[r+2], \ldots, \mathsf{LCP}[q]\}$$

holds for $0 \leq r < q < n$. The LCP array can be computed from the string y and the array SA in linear time (cf. [2]). The pair of arrays $(\mathsf{SA}, \mathsf{LCP})$ is commonly known as the *suffix array* of the string y.

For two strings u, v such that $|u| = |v| = m$ the Hamming distance $d(u,v)$ of u and v is defined as the number of differences between u and v. For sake of completeness we define $d(u,v) = \infty$ for strings u, v such that $|u| \neq |v|$.

3 Approximate String Matching

We consider approximate string matching by the well known procedure called *filtration* or *partitioning into exact matches* (cf. [9, 8]). Let $x \in \Sigma^*$ be a pattern of length m. We want to find occurrences of x in the text y with up to k mismatches under the Hamming distance. Partitioning into exact matches works as follows. We partition x into $q > k$ fragments $x_0, \dots x_{q-1} \in \Sigma^+$. We search the lists occurrences X_i of x_i. For each of the possibilities of choosing $q-k$ of the q fragments, we merge the respective lists of positions using the respective position

offsets. This provides us with $\binom{q}{q-k}$ candidate position lists. The union X of these merged lists is a superset of the positions of occurrences of x in y with up to k mismatches. We obtain the list of occurrences of x in y by filtering X using an online algorithm for testing if the candidate positions designate occurrences with at most k mismatches.

As an example consider a pattern x we partition into three fragments x_0, x_1 and x_2 for searching its occurrences with 1 mismatch. We have to consider three pairs of fragments: (x_0, x_1), (x_1, x_2) and (x_0, x_2). The first two combinations are easily found using an index for y. We need only search for the patterns x_0x_1 and x_1x_2. The third requires merging of lists in the conventional scheme. If we have an index supporting searching patterns with gaps however, merging is no longer necessary. Supporting the search of patterns with gaps is the purpose of the gapped suffix array.

4 The Gapped Suffix Array

4.1 Definitions

We define a generalisation of the notion of *lexicographical order* which we call (g_0, g_1)-*lexicographical order*, where $g_0, g_1 \in \mathbb{N}$. A string $u \in \Sigma^*$ is (g_0, g_1)-lexicographically smaller than a string $v \in \Sigma^*$

- if $u[0\,..\,g_0-1] \neq v[0\,..\,g_0-1]$ then iff $u < v$
- otherwise ($u[0\,..\,g_0-1] = v[0\,..\,g_0-1]$), if $\min(|u|, |v|) > g_0 + g_1$ then iff $u[0\,..\,g_0-1]u[g_0+g_1\,..\,|u|-1] < v[0\,..\,g_0-1]v[g_0+g_1\,..\,|v|-1]$
- otherwise ($u[0\,..\,g_0-1] = v[0\,..\,g_0-1], \min(|u|, |v|) \leq g_0 + g_1$), iff $|u| < |v|$

Informally the definition means that we compare u and v ignoring the presence of the letters in the position interval $[g_0, g_0 + g1)$, where we have to take some care about those strings which end inside the gap area. The (g_0, g_1)-lexicographical order is a total order on a set of strings such that each string has a different length. We define the (g_0, g_1)-*gapped suffix array* of y, which we denote by $(g_0, g_1) - \mathsf{gSA}$ (or shorter gSA, if the parameters g_0 and g_1 are clear from the context), as the array containing the starting positions of the non-empty suffixes of y in (g_0, g_1)-lexicographically ascending order. The (g_0, g_1)-*prefix* of the string $u \in \Sigma^*$, which we denote by $D(g_0, g_1, u)$, is defined as $u[0\,..\,g_0-1]$ if $|u| \leq g_0+g_1$ and as $u[0\,..\,g_0-1]u[g_0+g_1\,..\,|u|-1]$ if $|u| > g_0+g_1$. We define the (g_0, g_1)-gLCP array (we use the shorter notation gLCP, if the parameters g_0 and g_1 are clear from the context) for the string y based on its (g_0, g_1)-gSA array by

$$\mathsf{gLCP}[r] = \mathsf{lcp}(D(g_0, g_1, y[\mathsf{gSA}[r-1]\,..\,n-1]), D(g_0, g_1, y[\mathsf{gSA}[r]\,..\,n-1]))$$

for $r > 0$ and $\mathsf{gLCP}[r] = 0$ for $r = 0$.

4.2 Searching using the gapped suffix array

Assume we are given a query x of length $m > g_0 + g_1$ and we want to find all occurrences of patterns in $x[0\,..\,g_0-1]\Sigma^{g_1}x[g_0+g_1\,..\,m-1]$ in a text y using

the array (g_0, g_1)-gSA for y. The search method we use is analogous to the one we would use for searching an ungapped pattern using the array SA. The only major difference is that we suitably substitute the lexicographic order by the (g_0, g_1)-lexicographic order in the binary search for the interval of gapped suffix matching x. Thus the time required to report the *occ* gapped occurrences of x in y is $O((m - g_1) \log n + occ)$ if we do not use an adjoint (g_0, g_1)-gLCP array and $O((m - g_1) + \log n)$ if we do.

4.3 Computing the gapped suffix array

For the rest of the section assume we have fixed two natural numbers g_0 and g_1 and want to compute the arrays (g_0, g_1)-gSA (short gSA) and (g_0, g_1)-gLCP (short gLCP). We now show how to deduce the sorting in gSA in linear time $O(n)$ from the suffix array of y.

Let GRANK$[r]$ be defined as the number of ranks $r' < r$ such that LCP$[r] < g_0$. GRANK contains the ranks of factors of y with length up to g_0. The largest number we can find in GRANK is n. If GRANK[ISA$[i]$] $<$ GRANK[ISA$[j]$] for two positions i, j, then the suffix at position i is lexicographically and (g_0, g_1)-lexicographically smaller than the one at position j. Thus the order of the suffixes between SA and gSA can only differ if GRANK$[r]$ = GRANK$[q]$ for two ranks r and q. If GRANK$[r]$ = GRANK$[q]$, then we can determine the order of the respective gapped suffixes in gSA by checking ISA[SA$[r] + g_0 + g_1$] and ISA[SA$[q] + g_0 + g_1$], given that these two are defined. A problem occurs for such ranks r where SA$[r]+g_0+g_1 \geq n$ because the obtained value is not a valid position on y and thus ISA is not defined for it. According to the definition of the (g_0, g_1)-lexicographic order, this problem can be solved by sorting along the array HRANK given by

$$\text{HRANK}[r] = \begin{cases} \text{ISA}[\text{SA}[r] + g_0 + g_1] + g_0 + g_1 & \text{if } SA[r] + g0 + g1 < n \\ n - 1 - \text{SA}[r] & \text{otherwise} \end{cases}$$

The range of numbers found in the array HRANK is $[0, n+g_0+g_1-1]$, in particular the upper bound is $O(n)$. We can compute a representation of the array gSA by sorting the sequence of ranks $0, \ldots, n-1$ by the pair (GRANK$[r]$, HRANK$[r]$). This can be performed efficiently in linear time using a two stage radix sort, where we first sort by HRANK and then by GRANK. The concrete array gSA can then be obtained from this intermediate representation by mapping each rank on the suffix array to the respective position.

Theorem 1. *Given a string y of length n and its suffix sorting SA, the gapped suffix array (g_0, g_1)-gSA of y can be computed in linear time $O(n)$.*

4.4 Computing the gapped LCP array gLCP

We show how to compute the gapped LCP array gLCP from the suffix array and the array $(g_0, g_1) -$ gSA (short gSA) in linear time. We require a constant time solution of the range minimum query (RMQ) problem after linear time

preprocessing (see e.g. [3]). We can obtain the array $(g_0, g_1)-\mathsf{gLCP}$ (short gLCP) by setting

$$\mathsf{gLCP}[r] = \begin{cases} \mathsf{LCP}[r] \text{ if } \mathsf{LCP}[r] < g_0 \\ g_0 \quad \text{if } \max(\mathsf{gSA}[r] + g_0 + g_1, \mathsf{gSA}[r-1] + g_0 + g_1) \geq n \\ g_0 + l \quad \text{otherwise, where } l = \min(\mathsf{LCP}[p] + 1, \ldots, \mathsf{LCP}[q]) \text{ for} \\ \qquad p' = \mathsf{ISA}[\mathsf{gSA}[r-1] + g_0 + g_1] \\ \qquad q' = \mathsf{ISA}[\mathsf{gSA}[r] + g_0 + g_1] \\ \qquad p = \min(p', q') \text{ and } q = \max(p', q') \end{cases}$$

As every single step in the computation takes constant time and we have n steps, the runtime for computing gLCP is $O(n)$.

Theorem 2. *Given a string y of length n and its suffix sorting SA, the gapped LCP array (g_0, g_1)-gLCP of y can be computed in linear time $O(n)$.*

5 Representing the array gSA in reduced space

The uncompressed version of the gSA array requires $n\lceil \log n \rceil$ bits. The text however can be stored in $n\lceil \log |\Sigma| \rceil$ bits. In applications the size of the alphabet is fixed an small. Thus the space taken by the gSA will often be much larger than the space required for the text. The array SA is compressible (cf. [5]). Unfortunately, methods for compressing SA cannot be applied for compressing the array gSA, as the compression of SA requires the sorting of the suffixes according to the lexicographical order, which in general is not the same as the (g_0, g_1)-lexicographical order.

We provide a simple method for storing the array gSA using less than $n \log n$ bits space on average. Decoding the compressed representation of gSA will require the array SA. We limit our description to the aspects necessary for searching using the array gSA, i.e. our description allows accessing values in gSA corresponding to a provided query string. The more general case of accessing gSA for a given rank r without knowing a corresponding string can be facilitated using some additional succinct data structures. We omit the description for lack of space. We assume that the query string has a length of at least g_0. For shorter strings searching on the suffix array is sufficient. This may enumerate occurrences in a different order. However, this is not critical in most applications.

Let $R[r] = \{r' | \mathsf{HRANK}[r'] = r\}$. Each $R[r]$ is given as an interval of ranks. Observe that the sequences found in SA and gSA in each such interval are permutations of each other. On average we can expect each interval to have a size of $\frac{n}{|\Sigma|^{g_0}}$. Thus the permutation transforming gSA into SA can be stored using $\lceil \log |R[r]| \rceil$ bits per number for each interval r. Each interval $R[r]$ is assigned to a unique prefix $u(r) \in \Sigma^*$ of length at most g_0. The left bound $\mathsf{low}(r)$ and right bound $\mathsf{up}(r)$ of the interval $R[r]$ can be obtained by searching $u(r)$ on the suffix array. Knowing $\mathsf{low}(r)$ and $\mathsf{up}(r)$ we can compute the number of bits $b(r) = \lceil \log \mathsf{up}(r) - \mathsf{low}(r) + 1 \rceil$ used to store the numbers in the interval. Let L be defined by $L(r) = b(R^{-1}[r])$. L can be stored and indexed for rank queries via

a wavelet tree (cf. [4]) using $n\lceil\log\lceil\log n\rceil\rceil + o(n \log\log n)$ bits. Let $C[r]$ denote the permutation mapping the portion $R[r]$ of gSA to SA and let C_i denote the concatenation of all $C[r]$ such that $b(r) = i$. We can obtain $\mathsf{gSA}[r]$ for the query v as $C_{b(u^{-1}(v[0..g_0-1]))}[\mathsf{rank}_{b(u^{-1}(v[0..g_0-1]))}(r)]$ in time $O(\log\log n)$. The size of the data structure on average is $n(\log n - g_0 \log|\Sigma|) + n\log\log n + o(n\log\log n)$ bits. Using a space efficient wavelet tree data structure, the size is dominated by the first two terms in practice.

6 Conclusion

In this paper we have presented the gapped suffix array as a new efficient data structure for approximate matching under the Hamming distance. We obtained the same query time as for the conventional suffix array. The gapped suffix array can be derived in linear time from a text and its suffix sorting. Open problems include an improved query time independent of the text size, a succinct representation in $n\log\Sigma + o(n\log\Sigma)$ space and whether the gLCP array can be computed in linear time without using RMQ queries.

References

1. H.-J. Böckenhauer and D. Bongartz. *Algorithmic Aspects of Bioinformatics*. Springer, 2007.
2. M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007. 392 pages.
3. J. Fischer and V. Heun. A New Succinct Representation of RMQ-Information and Improvements in the Enhanced Suffix Array. In B. Chen, M. Paterson, and G. Zhang, editors, *ESCAPE*, volume 4614 of *LNCS*, pages 459–470. Springer, 2007.
4. R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 841–850, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.
5. R. Grossi and J. S. Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. *SIAM J. Comput.*, 35(2):378–407, 2005.
6. G. M. Landau and U. Vishkin. Efficient string matching with k mismatches. *Theor. Comput. Sci.*, 43:239–249, 1986.
7. G. M. Landau and U. Vishkin. Fast string matching with k differences. *J. Comput. Syst. Sci.*, 37(1):63–78, 1988.
8. G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.
9. G. Navarro, R. A. Baeza-Yates, E. Sutinen, and J. Tarhio. Indexing methods for approximate string matching. *IEEE Data Eng. Bull.*, 24(4):19–27, 2001.